Math

Report No. 454

A DESTRUCTIVE LIST COPYING ALGORITHM

by

Edward M. Reingold

June 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 454

A DESTRUCTIVE LIST COPYING ALGORITHM

by

Edward M. Reingold

June 1971

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

An efficient, non-recursive algorithm is given for copying any LISP-type list. In particular, the algorithm requires no storage other than the new nodes into which the list is to be copied, and no additional bits per node for marking; the algorithm runs in time proportional to the number of nodes in the list. The original list structure is destroyed as it is copied.

Key Words and Phrases

List copying, list traversal, garbage collection, LISP.

CR Categories

4.19, 4.49

Introduction

A list traversal is an orderly procedure for visiting each node of the list. Such traversals are primal to many list algorithms; for example, garbage collection and list searching are essentially traversal algorithms; copying a LISP-type list, which may be recursive, also requires a traversal algorithm, but with significant modifications. In this case, the difficulty arises because even though the nodes of the list can be copied during the traversal, the contents of those nodes may be pointers which are local to the old list. Generally, there are two solutions to this problem:

(a) The copying can be a simple linear displacement of the list in which the pointers are transformed by adding or subtracting a given constant. Usually linear displacement is not satisfactory.

(b) An additional data structure such as a table, stack. or queue can be introduced to store the correspondence between nodes in the original list and those in the copy; see [3, problem 10 of § 2.3.5]. This can necessitate a large amount of core which may not be available when storage is at a premium.

Both of the above approaches may require one or two additional bits per node for marking.

This paper presents an efficient, non-recursive algorithm for copying any LISP-type list. The algorithm requires no storage other than the new list nodes and uses no extra bits per node for marking; it runs in time proportional to the number of nodes in the list to be copied. The only disadvantage of the algorithm is that the original copy of the list is destroyed as it is copied.

The Algorithm

The algorithm copies the list structure pointed to by HEAD into the contiguous block of storage WORD(1), WORD(2), WORD(3), ... It is assumed that each node of the list consists of two fields, LLINK and RLINK, left and right pointers, respectively. It is also assumed that the test "is the word L is in the new area" is a valid, effective test; since the area into which the list is to be copied usually consists of contiguous memory locations, that test is a simple compare on the address of the word L.

The algorithm is essentially a generalization of the postorder traversal in binary trees (see [3, p. 316]), and is based on two distinct ideas. The first idea is a modification of the main trick in the garbage collection algorithm due to Schorr and Waite [4] (see 3 p. 417-418]): the use of the right links of the nodes in the original list to store pointers back up the list -- this facilitates the traversal. The second idea is due independently to Cheney [1] and Edwards [2] and is the use of the left links of the nodes in the original list to point to the corresponding nodes in the new copy of the list -- this facilitates the copying. A node N in the new copy of the list is initially an exact copy of the node in the original list which corresponds to N. As the list is traversed, the pointers in N are changed to point to the correct nodes in the new copy of the list. The algorithm is as follows:

Step 1 (Initialize)

I ← 0
L ← HEAD
LL ← Λ

Step 2 (Copy node)

I ← I + 1
LLINK(WORD(I)) ← LLINK(L)
RLINK(WORD(I)) ← RLINK(L)
LLINK(L) ← WORD(I)
RLINK(L) ← LL
LL ← L

Step 3 (Get next node to be copied)

If LLINK(LLINK(L)) is not an atom and LLINK(LLINK(LLINK(L))) is not in the new area, then set

L ← LLINK(LLINK(L))

and go back to Step 2.

If RLINK(LLINK(L)) is not an atom and LLINK(RLINK(LLINK(L))) is not in the new area, then set

L ← RLINK(LLINK(L))

and go back to Step 2.

Step 4 (Change links in the copy)

If LLINK(LLINK(L)) is not an atom then set

LLINK(LLINK(L)) ← LLINK(LLINK(LLINK(L))).

If RLINK(LLINK(L)) is not an atom then set

RLINK(LLINK(L)) ← LLINK(RLINK(LLINK(L))).

Step 5 (Back up list)

L ← RLINK(L)
LL ← L

Step 6 (Finished ?)

If L = Λ we are done, otherwise return to Step 3.

In the event that the lists to be copied are restricted to trees, the algorithm can be modified to restore the original tree by changing Steps 4 and 5. This gives a tree copying algorithm similar to the one alluded to in [3, solution to problem 21 of § 2.3.1]. These algorithms are better than the "usual" tree copying algorithm (see [3, p. 327]) which requires a stack for unthreaded trees.

Example

Figures 1 through 8 show the sequence of steps in the copying of a simple list structure. Figure 1 shows what the situation is after executing Step 1. Figure 2 shows the changes after executing Step 2 for the first time having copied the node pointed to by HEAD into WORD(2). Figure 3 shows the changes after Step 3 and then Step 2 have been executed, the node which is the left link of HEAD having been copied into WORD(2). Figure 4 shows how the links of WORD(2) have been "localized" to the new copy of the list in Step 4 and how the pointer back up the list has been followed in Step 5. Then in Step 6, since L is not null, the algorithm returns to Step 3 and copies the node pointed to by the right link of HEAD, as shown in Figure 5 and Figure 6. Figure 7 shows the algorithm having backed up the tree back to HEAD and Figure 8 shows the final configuration with the links of WORD(1) having been localized to the new copy of the list. The list has now been completely copied into locations WORD(1), WORD(2), and WORD(3).

Acknowledgements

The author is grateful to Mr. Russell Atkinson who coded and tested the algorithm on a PDP-11 at the Department of Computer Science of the University of Illinois at Urbana-Champaign, and to Mr. Ian Cunningham who suggested a simplification of the algorithm.

References

[1] Cheney, C. J., A nonrecursive list compacting algorithm, Comm. ACM 13, 11 (Nov. 1970), 677-678.

[2] Edwards, D. No known reference; see [3, problem 9 of § 2.3.5].

[3] Knuth, D. E., The Art of Computer Programming, Volume 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

[4] Schorr, H. and Waite, W. M., An efficient machine-independent procedure for garbage collection in various list structures, Comm. ACM 10, 8 (Aug. 1967), 501-506.

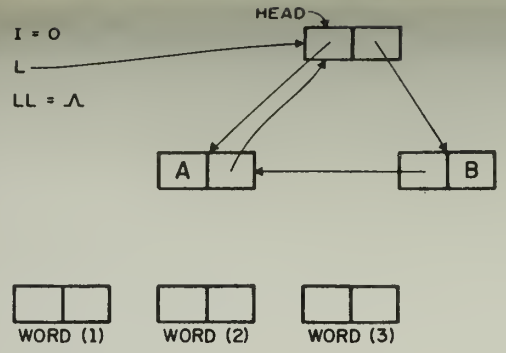

Fig. 1

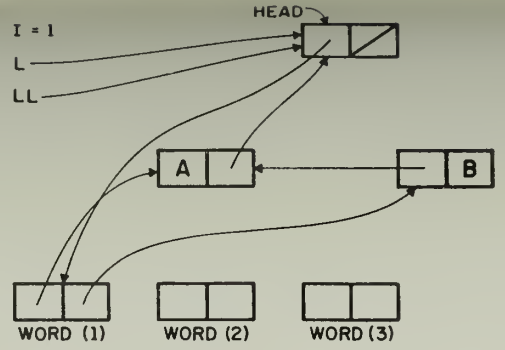

Fig. 2

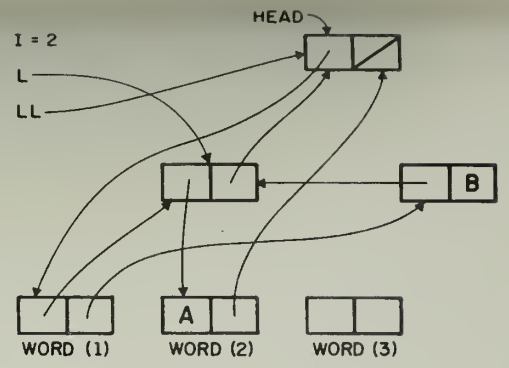

Fig. 3

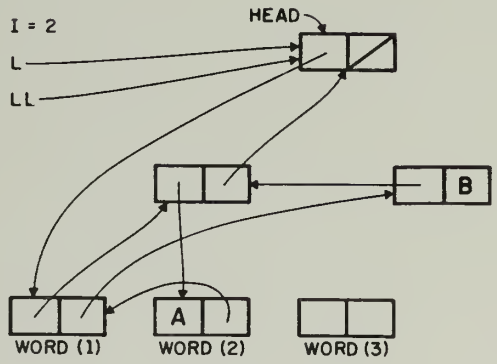

Fig. 4

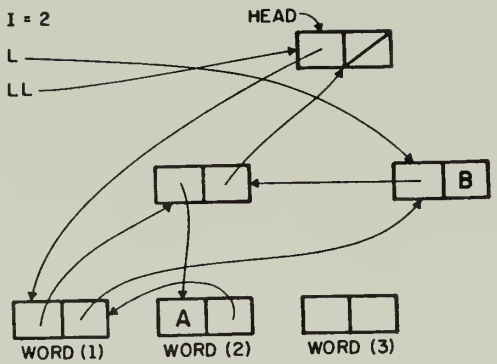

Fig. 5

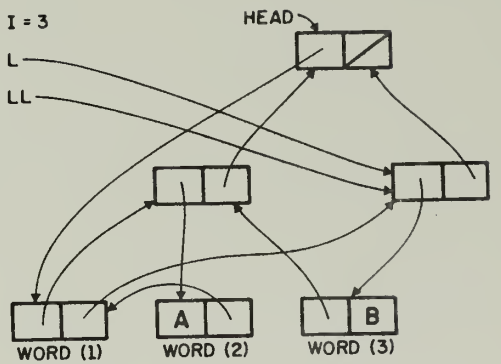

Fig. 6

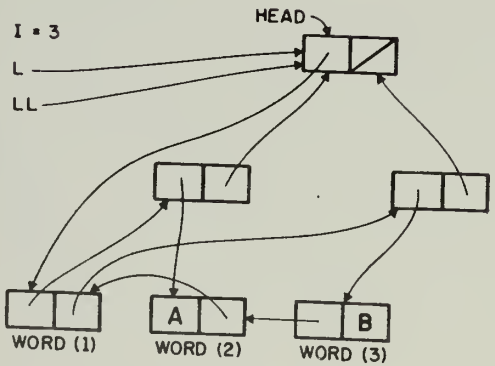

Fig. 7

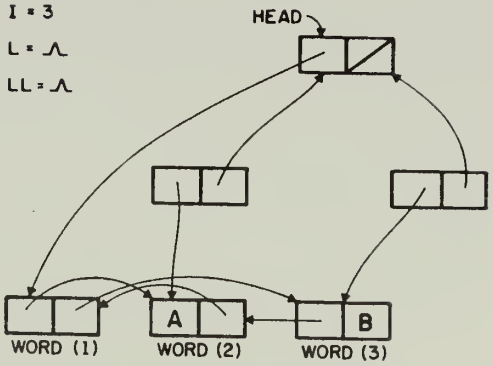

Fig. 8